A CryptoAPI Profile for One-Time Password Tokens

V1.0 Draft 3

RSA Security

August 17, 2005

Editor's note: This is the 2nd draft of this document, available for a 30-day public review period. Please send comments to: otps-editor@rsasecurity.com or to the otps mailing list: otps@rsasecurity.com

TABLE OF CONTENTS

1 Introduction

1.1 Scope

This document describes general Microsoft CryptoAPI ([1]) procedures and algorithms that can be used to retrieve and verify one-time passwords (OTPs) generated by OTP tokens. It also describes an OTP key import mechanism as well as a key generation mechanism that may be used to execute on-token OTP key generation.

1.2 Background

A One-Time Password token may be a handheld hardware device, a hardware device connected to a personal computer through an electronic interface such as USB, or a software module resident on a personal computer, which generates one-time passwords that may be used to authenticate a user towards some service. Increasingly, these tokens work in a connected fashion, enabling programmatic retrieval of their OTP values.

To meet the needs of applications wishing to access these connected OTP tokens in an interoperable manner, this document describes a method to support One-Time Password tokens in CryptoAPI v2.0 by defining a common OTP key type with an extensible set of attributes and by describing how CryptoAPI functions can be used to retrieve and verify OTP values generated by OTP tokens. It also describes an OTP key generation mechanism that may be used to execute on-token key generation.

Building on the OTP framework, the document specifies the RSA SecurID® OTP algorithm[1] and the OATH HOTP algorithm[2]. Additional algorithms may be defined separately to support other types of OTP tokens.

1.3 Document organization

The organization of this document is as follows:

- Section 1 is an introduction.
- Section 2 defines some notation used in this document.
- Section 3 defines a framework for recognizing CSPs supporting OTP algorithms and describes their usage.

[1] RSA SecurID® two-factor authentication is a symmetric authentication method which is patented by RSA Security. A user authenticates by submitting a one-time password (OTP), or PASSCODE value generated by an RSA SecurID token. The RSA SecurID token may be a handheld hardware device, a hardware device connected to a personal computer through an electronic interface such as USB, or a software module resident on the personal computer

[2] The HOTP algorithm is work in progress, currently defined in the IETF draft http://www.ietf.org/internet-drafts/draft-mraihi-oath-hmac-otp-04.txt developed by the Open Authentication initiative (http://www.openauthentication.org)

- Section 4 uses the framework to define specific OTP algorithm types and describe their usage.
- Appendix A collects the definition of constants defined herein.
- Appendix B contains example usage.
- Appendices C, D and E cover intellectual property issues, give references to other publications and standards, and provide general information about the One-Time Password Specifications.

2 Acronyms and definitions

2.1 Acronyms

CSP	Cryptographic Service Provider
OTP	One-Time Password

2.2 Definitions

For the purposes of this document, the following definitions apply:

Cryptographic Device	A device storing OTP keys and possibly performing cryptographic functions.
Token	A logical cryptographic device.

3 CSP definitions and usage

3.1 Introduction

An OTP CSP manages OTP keys and allows applications to retrieve and validate OTP codes. This section defines the identification, usage, and external behavior of such a CSP.

3.2 CSP identification

The type of a CSP that supports the algorithms and procedures described herein shall be as follows:

```
PROV_OTP
```

Vendors of CSPs of type **PROV_OTP** will use vendor-specific CSP names.

3.3 OTP algorithm identification

All OTP algorithms shall be of the type **ALG_TYPE_OTP**. All OTP generation shall be represented as a keyed hash algorithm of type **CALG_OTP** with the actual algorithm used being represented by the type of the secret key.

As with standard keys, OTP keys may be stored in a key container but there may only be one key of each type in such a container.

3.4 CSP Usage

3.4.1 Initialization

The application selects an OTP-supporting CSP and optionally a suitable key container if there are several. If several tokens supported by the same CSP are connected, it is up to the provider to choose the proper token based on passed container name.

3.4.2 OTP Generation

To generate actual OTPs, the application proceeds as follows:

1. Call **CryptAcquireContext** to get a handle to a key container containing the required keys. The name of the key container may be given in the *pszContainer* parameter in order to explicitly specify the container to use. If the container name is omitted then the default container shall be used. Selection of the default container is CSP specific.
2. Optionally call **CryptGetUserKey** to obtain a handle to the desired OTP key.

hProv	Shall be set to the provider handle.
AlgID	Either set to an explicit OTP algorithm identifier or to **AT_OTP** if the default key is wanted.

3. Optionally call **CryptGetKeyParam** to retrieve key parameters of the key giving properties of the key.
4. Call **CryptCreateHash** to create an OTP hash object to generate the OTP.

hProv	Shall be set to the provider handle.
AlgID	Shall be set to **CALG_OTP**.
hKey	If this parameter is NULL then the default OTP key of the specified container will be used otherwise the specified OTP key will be used.
dwFlags	Shall be set to 0.

5. Optionally call **CryptSetHashParam** to set the values to be used in the OTP calculation associated with the selected OTP key.

hHash	Shall be set to the hash handle created above.
dwParam	Shall be set to **HP_OTP_PARAMS**.
pbData	Pointer to an **OTP_PARAMETERS** structure containing the values.
dwFlags	Shall be set to 0.

6. Call **CryptHashData**.

pbData	Shall be set to NULL.

dwDataLen	Shall be set to 0.
dwFlags	This parameter may contain flag values specific to the OTP algorithm.

7. Call **CryptGetHashParam.**

dwParam	Shall be set to **HP_HASHVAL**.
pbData	Shall point to a buffer area that will receive the OTP value.
pdwDataLen	Indicates the size of this data buffer.

8. Optionally call **CryptGetHashParam** to retrieve parameter values associated with the generated OTP.

   Shall be set to the hash handle created above.

dwParam	Shall be set to **HP_OTP_PARAMS**.
pbData	Pointer to an **OTP_PARAMETERS** structure that will receive the values.
dwFlags	Shall be set to 0.

3.4.3 OTP Validation

To validate a given OTP, an application proceeds as described above and compares the given and generated OTP codes.

3.4.4 Provider Parameters

Providers of type **PROV_OTP** will support the following additional parameters.

Value	Meaning
PP_OTP_ALG	AlgID of OTP algorithm mapped to **AT_OTP**.
PP_OTP_PINPAD	A **BOOL** indicating whether the device has a PIN-pad.

Table 1: Common OTP Provider Parameters

3.4.5 Key Management

3.4.5.1 Key Containers

As with any key, OTP keys will be contained within key containers with any single container containing at most one OTP key of a given algorithm. A provider may associate one of these keys with **AT_OTP**.

Applications may assign a label to a key container to contain the keys about to be created by setting the *pszContainer* parameter of **CryptAcquireContext**, when the *dwFlags* parameter is set to **CRYPT_NEW_KEYSET**.

3.4.5.2 Importing Keys

If supported by the CSP, OTP keys shall be imported using **CryptImportKey** either in plain form as a **PLAINTEXTBLOB** or wrapped in a **SIMPLEBLOB**. The blob content is the value of OTP key with the **ALG_ID** of the **BLOBHEADER** set to the OTP algorithm.

Once the key has been imported, it will be configured by setting key parameters using **CryptSetKeyParam** but the key will not be available for use until **KP_ALGID** has been set to the same value as contained in the **BLOBHEADER**. The setting of **KP_ALGID** is therefore mandatory and will signal to the provider that the generation of the key is complete.

3.4.5.3 Exporting Keys

If supported by the CSP, OTP keys shall be exported using **CryptExportKey** in wrapped form as a **SIMPLEBLOB** or unwrapped as a **PLAINTEXTBLOB**.

3.4.5.4 Generating Keys

The **CryptGenKey** function shall be called with the *Algid* parameter set to the OTP algorithm or to **AT_OTP** to generate a key of the default type.

Once the key has been generated, it will be configured by setting key parameters using **CryptSetKeyParam** but the key will not be available for use until **KP_ALGID** has been set to the same value as the algorithm of the key being generated. The setting of **KP_ALGID** is therefore mandatory and will signal to the provider that the generation of the key is complete.

3.4.5.5 Getting handle to existing key

The handle to the existing OTP key is obtained using **CryptGetUserKey** with the *dwKeySpec* parameter set to the required algorithm identifier.

Alternatively, a handle for the default OTP key of the key container may be obtained using the **AT_OTP** key identifier.

3.4.6 Key parameters

OTP keys may have various parameters such as serial number, service identifier etc. Generally speaking these parameters are available via **CryptGetKeyParam** function and depending on provider policy, may be changed via **CryptSetKeyParam** function. However, vendors may restrict the ability of applications to change key attributes after the key has been created.

The following parameters are defined for all OTP keys.

Value	Meaning
KP_OTP_END_DATE	A **SYSTEMTIME** giving the expiry date of the key.
KP_OTP_SERIAL_NUMBER	A **BYTE** array containing an issuer specific serial number of key.

	If being set using **CryptSetKeyParam** then the array must be passed in a **CRYPTOAPI_BLOB**. If being retrieved using **CryptGetKeyParam** then the array will be returned directly.
KP_OTP_FORMAT	A **DWORD** giving the format of OTP values produced with this key. This can be one of: CRYPT_OTP_FORMAT_DECIMAL CRYPT_OTP_FORMAT_HEXADECIMAL CRYPT_OTP_FORMAT_ALPHANUMERIC
KP_OTP_LENGTH	A **DWORD** value giving the default length of OTP values produced by the key.
KP_OTP_COUNTER	A **BYTE** array containing the current value of the counter for the key.
KP_OTP_TIME	A **SYSTEMTIME** giving the keys time value.
KP_OTP_USER_IDENTIFIER	A UTF8 encoded **CHAR** string that identifies a user associated with the OTP key (may be used to enhance the user experience).
KP_OTP_SERVICE_IDENTIFIER	An identifier in the form of a null-terminated UTF8 encoded **CHAR** string of a service that may validate OTPs generated by this key.
KP_OTP_SERVICE_LOGO	Logotype image that identifies a service that may validate OTPs generated by this key.
KP_OTP_SERVICE_LOGO_TYPE	MIME type of the **KP_OTP_SERVICE_LOGO** attribute value as a UTF encoded **CHAR** string.
KP_OTP_ID	A **BYTE** array containing a globally unique identifier of the key. The value for the parameter will be initialized by the provider whenever a key is created but may be modified once the object is created.
KP_OTP_LENGTH_MIN	A **DWORD** value giving the minimum length of OTP values supported by the key or mandated by security policy.
KP_OTP_LENGTH_MAX	A **DWORD** value giving the maximum length of OTP values supported by the key or mandated by security policy.

Table 2: Common OTP Key Parameters

Calculations of OTPs can be based on different types of information. The following parameters define the information that is or may be used by a particular key when generating an OTP value. The actual values used in the calculation are taken from the corresponding OTP parameter.

Value	**Meaning**
KP_OTP_PIN_REQUIREMENT	Requirements on providing a PIN.
KP_OTP_CHALLENGE_REQUIREMENT	Requirements on providing a challenge.
KP_OTP_TIME_REQUIREMENT	Requirements on providing a time value.

KP_OTP_COUNTER_REQUIREMENT	Requirements on providing a counter value.

Table 3: Common OTP Requirement Key Parameters

The above mode parameters are all **DWORD** values that can contain one of the following values.

OTP Calculation Mode	Description
CRYPT_OTP_VALUE_NOT_USED	The value will be ignored if it is provided.
CRYPT_OTP_VALUE_NOT_REQUIRED	The provider will provide a value if it is not given by the caller.
CRYPT_OTP_VALUE_REQUIRED	Value must be provided in order to carry out the OTP calculation.

Table 4: OTP Requirement Key Parameter Values

3.4.7 Hash Parameters

As described above, an OTP is generated using **CryptHashData** on an OTP hash object. The parameters used to generate the OTP will depend on the algorithm of the key used and will be given by the requirements attributes of that key as shown in Table 3.

This information is passed to the provider using **CryptSetHashParam** to set the **HP_OTP_PARAMS** hash parameter with the *pbData* value set to a pointer to an **OTP_PARAMETERS** structure. This structure contains an array of **OTP_PARAMETER** structures containing the values to be set.

This structure is defined as follows:

```
typedef struct _OTP_PARAMETERS
{
    DWORD cParams;
    POTP_PARAMETER pParams;
} OTP_PARAMETERS, *POTP_PARAMETERS;
```

The fields of the structure have the following meanings:

cParam	The number of elements in the *pParams* array.
pParams	Array of **OTP_PARAMETER** structures

Each element in the parameters array will be one of the following structures.

```
typedef struct _OTP_PARAMETER
{
    DWORD paramID;
    CRYPT_DATA_BLOB paramValue;
} OTP_PARAMETER, *POTP_PARAMETER;
```

The fields in the above structure have the following meaning:

paramID	The identifier of the type of data contained in the *paramValue* element.

paramValue The value of the parameter. The *cbData* member of the **CRYPT_DATA_BLOB** structure indicates the length of the *pbData* member. The *pbData* member contains the attribute information.

The actual values used to generate an OTP can be retrieved by an application using **CryptGetHashParam** to retrieve the **HP_OTP_PARAMS** hash parameter. The returned information will be stored contiguously in the returned buffer with all pointers set to point within the allocated block of memory. As with any other hash parameter, the caller should first call with *pbData* set to NULL to determine the size of the required buffer.

The following parameters are defined.

Value	**Meaning**
CRYPT_OTP_PIN	A UTF8 encoded **CHAR** string containing the PIN to use in the OTP calculation. Setting of this parameter is controlled by **KP_OTP_PIN_REQUIREMENT**.
CRYPT_OTP_CHALLENGE	A **BYTE** array containing a challenge to use in the OTP calculation. Setting of this parameter is controlled by **KP_OTP_CHALLENGE_REQUIREMENT**.
CRYPT_OTP_TIME	A **SYSTEMTIME** giving the time to use in the OTP calculation. Setting of this parameter is controlled by **KP_OTP_TIME_REQUIREMENT**.
CRYPT_OTP_COUNTER	A **BYTE** array containing the current value of the counter for the key. Setting of this parameter is controlled by **KP_OTP_COUNTER_REQUIREMENT**.
CRYPT_OTP_FLAGS	A **DWORD** containing bit flags indicating the characteristics of the sought OTP as defined below.
CRYPT_OTP_FORMAT	A **DWORD** giving the desired format of OTP values produced.
CRYPT_OTP_LENGTH	A **DWORD** giving the desired OTP length.

Table 5: Common OTP Parameters

The parameters **CRYPT_OTP_PIN**, **CRYPT_OTP_CHALLENGE**, **CRYPT_OTP_TIME** and **CRYPT_OTP_COUNTER** correspond to OTP key parameters in Table 3 and are used to set the value to be used in the OTP calculation.

- If the key parameter has the value **CRYPT_OTP_PARAM_REQUIRED** then the corresponding hash parameter must be set before the OTP is calculated.
- If the key parameter has the value **CRYPT_OTP_PARAM_NOT_REQUIRED** then the corresponding hash parameter may be set. If a value is not provided then the provider may collect it during the call to **CryptHashData**.

- If the key parameter has the value **CRYPT_OTP_PARAM_NOT_USED** then any value assigned to the corresponding hash parameter will be ignored.

The following table defines the flag values for **CRYPT_OTP_FLAGS**.

Flag	Meaning
CRYPT_OTP_NEXT_OTP	Set if the hash shall return the next OTP rather than the current one.
CRYPT_OTP_NO_PIN	Set if the OTP calculation shall not include the PIN value.
CRYPT_OTP_NO_CHALLENGE	Set if the OTP calculation shall not include the challenge value.
CRYPT_OTP_NO_TIME	Set if the OTP calculation shall not include the time value.
CRYPT_OTP_NO_COUNTER	Set if the OTP calculation shall not include the counter value.

Table 6: Common OTP Flags

4 OTP algorithm definitions

4.1 RSA SecurID

4.1.1 Algorithm Identification

The RSA SecurID algorithm is defined as:

```
CALG_SECURID
```

4.1.2 Key Parameters

These keys support the following key parameters in addition to those in Table 3.

Value	Meaning
KP_OTP_TIME_INTERVAL	A **DWORD** containing the interval, in seconds, between OTP values produced by the key.

Table 7: SecurID OTP Key Parameters

4.1.3 OTP Parameters

SecurID OTP values are not counter based and so any value assigned to **CRYPT_OTP_COUNTER** will be ignored.

4.2 HOTP

4.2.1 Algorithm Identification

The HOTP algorithm is defined as:

```
CALG_HOTP
```

4.2.2 Key Parameters

For HOTP keys the **KP_OTP_COUNTER** and **CRYPT_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form.

The **KP_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this parameter may not be modified and/or may not be revealed

4.2.3 OTP Parameters

HOTP OTP values are not time based and so any value assigned to **CRYPT_OTP_TIME** will be ignored.

A. Manifest constants

A.1 Provider Type

```
#define PROV_OTP               0xTBD
```

A.2 Algorithm Identifiers

```
#define ALG_TYPE_OTP           0xTBD

#define ALG_SID_OTP_HMAC       0xTBD
#define ALG_SID_OTP_SECURID    0xTBD
#define ALG_SID_OTP_HOTP       0xTBD

#define CALG_OTP \
  (ALG_CLASS_HASH | ALG_TYPE_OTP | ALG_SID_OTP_HMAC)
#define CALG_SECURID \
  (ALG_CLASS_HASH | ALG_TYPE_OTP | ALG_SID_OTP_SECURID)
#define CALG_HOTP \
  (ALG_CLASS_HASH | ALG_TYPE_OTP | ALG_SID_OTP_HOTP)

#define AT_OTP                TBD
```

A.3 OTP Parameters

```
#define CRYPT_OTP_PIN          1
#define CRYPT_OTP_CHALLENGE    2
#define CRYPT_OTP_TIME         3
#define CRYPT_OTP_COUNTER      4
#define CRYPT_OTP_FLAGS        5
#define CRYPT_OTP_FORMAT       6
#define CRYPT_OTP_LENGTH       7
```

A.4 OTP Flags

```
#define CRYPT_OTP_NEXT_OTP      0x00000001
#define CRYPT_OTP_NO_TIME       0x00000002
#define CRYPT_OTP_NO_COUNTER    0x00000004
#define CRYPT_OTP_NO_CHALLENGE  0x00000008
#define CRYPT_OTP_NO_PIN        0x00000010
```

A.5 Key Parameters

```
#define KP_OTP_PIN_REQUIREMENT       0xTBD
#define KP_OTP_CHALLENGE_REQUIREMENT 0xTBD
#define KP_OTP_TIME_REQUIREMENT      0xTBD
#define KP_OTP_COUNTER_REQUIREMENT   0xTBD

#define KP_OTP_END_DATE              0xTBD
#define KP_OTP_SERIAL_NUMBER         0xTBD
#define KP_OTP_FORMAT                0xTBD
#define KP_OTP_LENGTH                0xTBD
#define KP_OTP_COUNTER               0xTBD
#define KP_OTP_TIME                  0xTBD
#define KP_OTP_USER_IDENTIFIER       0xTBD
#define KP_OTP_SERVICE_IDENTIFIER    0xTBD
#define KP_OTP_SERVICE_LOGO          0xTBD
#define KP_OTP_SERVICE_LOGO_TYPE     0xTBD
#define KP_OTP_ID                    0xTBD
#define KP_OTP_LENGTH_MIN            0xTBD
#define KP_OTP_LENGTH_MAX            0xTBD

#define KP_OTP_TIME_INTERVAL         0xTBD
```

A.6 Key Parameter Constants

```
#define CRYPT_OTP_PARAM_NOT_USED          0
#define CRYPT_OTP_PARAM_NOT_REQUIRED      1
#define CRYPT_OTP_PARAM_REQUIRED          2

#define CRYPT_OTP_FORMAT_DECIMAL          0
#define CRYPT_OTP_FORMAT_HEXADECIMAL      1
#define CRYPT_OTP_FORMAT_ALPHANUMERIC     2
```

A.7 Hash Parameters

```
#define HP_OTP_PARAMS                0xTBD
```

A.8 Provider Parameters

```
#define PP_OTP_ALGID                 0xTBD
#define PP_OTP_PINPAD                0xTBD
```

B. Examples Using OTP Keys

B.1 OTP Retrieval

The following code sample illustrates the retrieval of an OTP value from an RSA SecurID token using **CryptHashData**.

```
HCRYPTPROV hProv = (HCRYPTPROV)NULL;
HCRYPTHASH hHash = (HCRYPTHASH)NULL;
HCRYPTKEY hKey = (HCRYPTKEY)NULL;

DWORD pinReq;
DWORD pinReqLen = sizeof(pinReq);

CHAR *pPIN = "...";

DWORD len = 0;
DWORD OTPLen = 0;
BYTE *pbOTP = NULL;

DWORD ret = ERROR_GEN_FAILURE;

__try
{
    /*  Acquire context to default container of default
        OTP provider. */
    if (!CryptAcquireContext(&hProv, NULL, NULL,
        PROV_OTP, 0))
    {
        ret = GetLastError();
        printf("Error: Failed to open key
        container.\n");
        __leave;
    }

    /* Get a handle to the default OTP key of the
        container. */
    if (!CryptGetUserKey(hProv, AT_OTP, &hKey))
    {
        ret = GetLastError();
        printf("Error: Unable to find OTP key in
        container.\n");
        __leave;
    }

    /* Create the OTP hash object. */
    if (!CryptCreateHash(hProv, CALG_OTP, hKey, 0,
        &hHash))
    {
        ret = GetLastError();
        printf("Error: Failed to create the OTP hash
        object.\n");
        __leave;
    }

    /* Find out the PIN requirements of the key. */
    if (!CryptGetKeyParam(hKey, KP_OTP_PIN_REQUIREMENT,
        (BYTE *)&pinReq, &pinReqLen, 0))
    {
        ret = GetLastError();
        printf("Error: Failed to retrieve
        KP_OTP_PIN_REQUIREMENTS.\n");
        __leave;
    }

    /* Set the PIN if one is required. */
    if ((pinReq == CRYPT_OTP_PARAM_REQUIRED) ||
        (pinReq == CRYPT_OTP_PARAM_NOT_REQUIRED))
    {
        OTP_PARAMETER pinParam;
        OTP_PARAMETERS otpParams;

        pinParam.paramID = CRYPT_OTP_PIN;
        pinParam.paramValue.cbData = lstrlen(pPIN);
        pinParam.paramValue.pbData = (BYTE *)pPIN;

        otpParams.cParams = 1;
        otpParams.pParams = &pinParam;

        if (!CryptSetHashParam(hHash, HP_OTP_PARAMS,
        (const BYTE *)&otpParams, 0))
        {
            ret = GetLastError();
            printf("Error: Failed to set OTP PIN.\n");
            __leave;
        }
    }

    /* Generate the OTP. */
    if (!CryptHashData(hHash, NULL, 0, 0))
    {
        ret = GetLastError();
        printf("Error: Failed to generate OTP.\n");
        __leave;
    }

    /* Retrieve the length of the OTP */
```

```
        len = sizeof(OTPLen);
        if(!CryptGetHashParam(hHash, HP_HASHSIZE, (BYTE
            *)&OTPLen, &len, 0))
        {
            printf("Failed to retrieve length of OTP.\n");
            __leave;
        }

        /* Retrieve the OTP */
        pbOTP = (BYTE*)malloc(OTPLen);
        if (!pbOTP)
        {
            printf("Error: Failed to allocate buffer for
            OTP..\n");
            ret = ERROR_OUTOFMEMORY;
            __leave;
        }

        len = OTPLen;
        if (!CryptGetHashParam(hHash, HP_HASHVAL, pbOTP,
            &len, 0))
        {
            ret = GetLastError();
            printf("Error: Failed to retrieve OTP
            value.\n");
            __leave;
        }

        /* Use the returned pbOTP */

        ret = ERROR_SUCCESS;
    }
    __finally
    {
        if (hHash)
            CryptDestroyHash(hHash);

        if (hKey)
            CryptDestroyKey(hKey);

        if (hProv)
            CryptReleaseContext(hProv, 0);

        If (pbOTP)
            free(pbOTP);
    }

    return ret;
```

C. Intellectual property considerations

RSA Security makes no patent claims on the general constructions described in this document, although specific underlying techniques may be covered. The RSA SecurID technology is covered by a number of US patents (and foreign counterparts), in particular US patent nos. 4,720,860, 4,856,062, 4,885,778, 5,097,505, 5,168,520, and 5,657,388. Additional patents are pending.

D. References

[1] Microsoft Corporation. *Microsoft Cryptographic API v2.0*. URL: http://msdn.microsoft.com/library/en-us/seccrypto/security/cryptography_portal.asp

E. About OTPS

The *One-Time Password Specifications* are documents produced by RSA Security in cooperation with secure systems developers for the purpose of simplifying integration and management of strong authentication technology into secure applications, and to enhance the user experience of this technology.

RSA Security plans further development of the OTPS series through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. As four our PKCS documents, results may also be submitted to standards forums. For more information, contact:

OTPS Editor
RSA Security
174 Middlesex Turnpike
Bedford, MA 01730 USA
otps-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/